SIAM REVIEW
Vol. 44, No. 3, pp. 373–393

Effects of Ordering Strategies and Programming Paradigms on Sparse Matrix Computations*

Leonid Oliker[†]
Xiaoye Li[†]
Parry Husbands[†]
Rupak Biswas[‡]

Abstract. The conjugate gradient (CG) algorithm is perhaps the best-known iterative technique for solving sparse linear systems that are symmetric and positive definite. For systems that are ill conditioned, it is often necessary to use a preconditioning technique. In this paper, we investigate the effects of various ordering and partitioning strategies on the performance of parallel CG and ILU(0) preconditioned CG (PCG) using different programming paradigms and architectures. Results show that for this class of applications, ordering significantly improves overall performance on both distributed and distributed shared-memory systems, cache reuse may be more important than reducing communication, it is possible to achieve message-passing performance using shared-memory constructs through careful data ordering and distribution, and a hybrid MPI+OpenMP paradigm increases programming complexity with little performance gain. A multithreaded implementation of CG on the Cray MTA does not require special ordering or partitioning to obtain high efficiency and scalability, giving it a distinct advantage for adaptive applications; however, it shows limited scalability for PCG due to a lack of thread-level parallelism.

Key words. preconditioned conjugate gradient, graph partitioning, reverse Cuthill–McKee, self-avoiding walks, message passing, shared-memory directives, hybrid programming, multithreading

AMS subject classifications. 65Y05, 68W10, 65Y10, 65F50, 65F10

PII. S0036144500382076

1. Introduction.

The ability of computers to solve hitherto intractable problems and simulate complex processes using mathematical models makes them an indispensable part of modern science and engineering. Computer simulations of large-scale realistic applications usually require solving a set of nonlinear partial differential equations (PDEs) over a finite region, subject to certain initial and boundary conditions. Structured grids are the most natural way to discretize such a computational domain since they are characterized by a uniform connectivity pattern. Their regular structure is

*Received by the editors December 6, 2000; accepted for publication (in revised form) February 20, 2002; published electronically August 1, 2002.
http://www.siam.org/journals/sirev/44-3/38207.html

[†]National Energy Research Scientific Computing Center (NERSC), Mail Stop 50F, Lawrence Berkeley National Laboratory, Berkeley, CA 94720 (loliker@lbl.gov, xsli@lbl.gov, pjrhusbands@lbl.gov). These authors were supported by the Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract DE-AC03-76SF00098.

[‡]NASA Advanced Supercomputing (NAS) Division, NASA Ames Research Center, Mail Stop T27A-1, Moffett Field, CA 94035-1000 (rbiswas@nas.nasa.gov).

also well suited for simple ordering techniques. Unfortunately, complicated domains must often be divided into multiple structured grids to be completely discretized, requiring a great deal of human intervention. Unstructured meshes, by contrast, can be generated automatically for applications with complex geometries or those with dynamically moving boundaries, but at the cost of higher memory requirements to explicitly store the connectivity information for every point in the mesh. However, because such meshes are irregularly structured, sophisticated ordering schemes are required to achieve high performance on leading parallel systems. In this paper, we examine the relationship between the ordering of unstructured meshes and the corresponding parallel performance of the underlying numerical solution.

Using a standard fixed mesh, it may be time consuming or even impossible for a simulation to resolve fine-scale features. Efficiency can be significantly improved by locally refining and coarsening the mesh to capture the phenomena of interest. Briefly, the existing grid is modified by inserting new points in regions that require more resolution and removing points from regions where less resolution is acceptable. Unstructured grids, by their very nature, facilitate this kind of local dynamic mesh adaptation to efficiently solve problems with evolving physical features such as shock waves, vortices, detonations, shear layers, and crack propagation.

The process of obtaining numerical solutions to the governing PDEs requires solving large sparse linear systems or eigensystems defined over the unstructured meshes that model the underlying physical objects. The conjugate gradient (CG) algorithm is perhaps the best-known iterative technique for solving sparse linear systems that are symmetric and positive definite. The CG algorithm is often used with a preconditioner for systems that are ill conditioned. When using incomplete factorization as a preconditioner, the sparse matrix–vector multiply (SPMV) and the triangular solves are usually the most expensive operations within each iteration of preconditioned CG (PCG).

On uniprocessor machines, numerical solutions of such complex, real-life problems can be extremely time consuming, a fact driving the development of increasingly powerful multiprocessor supercomputers. The unstructured, dynamic nature of many systems worth simulating, however, makes their efficient parallel implementation a daunting task. This is primarily due to the load imbalance created by the dynamically changing nonuniform grids and the irregular data access patterns [14, 15, 21]. These, in turn, leave many processors idle and cause significant unbalanced communication at runtime, thereby adversely affecting the total execution time.

Furthermore, modern computer architectures, based on deep memory hierarchies, show acceptable performance for irregular computations only if users care about the proper distribution and placement of their data [2, 13]. Single-processor performance depends crucially on the exploitation of locality, and parallel performance degrades significantly if inadequate partitioning of data causes excessive communication and/or data migration. An intuitive approach would be to use a sophisticated partitioning algorithm, and then to postprocess the resulting partitions with an enumeration strategy for enhanced locality. Although, in that sense, optimizations for partitioning and locality may be treated as separate problems, real applications tend to show a rather intricate interplay of both.

In this paper, we investigate the effects of various ordering and partitioning strategies on the performance of CG and PCG using different programming paradigms and architectures. In particular, we use the reverse Cuthill–McKee [4] and the self-avoiding walks [7] ordering strategies, and the METIS [12] graph partitioner. We examine parallel implementations of CG and PCG using MPI, shared-memory compiler directives, hybrid programming (MPI+OpenMP), and fine-grained multithreading on four state-of-the-art parallel supercomputers: a Cray T3E, an SGI Origin2000, an IBM SP, and a Cray (formerly Tera) MTA. Results show that for this class of applications, ordering significantly improves overall performance, cache reuse may be more important than reducing communication, it is possible to achieve message-passing performance using shared-memory constructs through careful data ordering and distribution, and the hybrid paradigm increases programming complexity with little performance gain. However, the multithreaded implementation of CG does not require special ordering or partitioning to obtain high efficiency and scalability, giving it a distinct advantage for adaptive applications even though it shows limited scalability for PCG due to a lack of thread-level parallelism.

The remainder of the paper is organized as follows. In section 2, we give a brief overview of CG and ILU(0) preconditioning. The partitioning and ordering strategies are described in section 3. In section 4, we describe the various parallel machines and the corresponding programming paradigms used for our experiments. Detailed performance results are presented in section 5. Finally, section 6 concludes the paper with a summary and some observations.

2. Sparse Matrix Computations.

A discretization of a PDE typically leads to large sparse matrices, which are commonly defined as matrices that have very few nonzero entries. Special sparse matrix solution techniques can be used whenever the zero elements need not be stored. Direct solution methods were traditionally preferred because of their robustness and predictable nature. However, iterative algorithms are now becoming quite popular, especially for large problems.

The earliest iterative methods used a relaxation technique in which the components of the approximation were systematically modified until convergence. This class consists of Jacobi, Gauss–Seidel, and the alternating direction implicit (ADI) algorithms. A second group of iterative techniques uses a projection process, which is a canonical way of extracting an approximate solution from a subspace. The steepest descent and minimal residual schemes belong to this class. However, other projection-based iterative techniques that use Krylov subspaces are currently considered to be among the most important for solving large sparse matrices. The CG algorithm is perhaps the best known in this class. Although theoretically robust, CG can suffer from slow convergence for ill-conditioned matrices. It is therefore common to use a preconditioning technique with CG. In the following subsections, we briefly describe the preconditioned CG algorithm and the incomplete LU (ILU) preconditioner.

2.1. Conjugate Gradient.

The CG algorithm is the oldest and best-known Krylov subspace method used to solve the sparse symmetric positive definite linear system $Ax = b$. The method starts from an initial guess x_0 of the vector x. It then successively generates approximate solutions in the Krylov subspace and search directions used in updating the approximate solution and residual.

The convergence rate of CG depends on the spectral condition number of the coefficient matrix A. For ill-conditioned linear systems, it is often necessary to use a preconditioning technique. In other words, the original system is transformed into another that has the same solution, but with better spectral properties. For instance, if matrix M approximates A in some sense, then $M^{-1}Ax = M^{-1}b$ would have a better condition number. The PCG algorithm [19] is outlined in Figure 2.1.

Each iteration of PCG involves one SPMV for Ap_j, one solve with preconditioner M, three vector updates (AXPY) for x_{j+1}, r_{j+1}, and p_{j+1}, and three inner products (DOT) for the update scalars α_j and β_j, which make the generated sequences

```
For an initial guess $x_0$, compute $r_0 = b - Ax_0$, $p_0 = z_0 = M^{-1}r_0$
for $j = 0, 1, \ldots$, until convergence
        $\alpha_j = (r_j, z_j)/(Ap_j, p_j)$
        $x_{j+1} = x_j + \alpha_j p_j$
        $r_{j+1} = r_j - \alpha_j Ap_j$
        $z_{j+1} = M^{-1}r_{j+1}$
        $\beta_j = (r_{j+1}, z_{j+1})/(r_j, z_j)$
        $p_{j+1} = z_{j+1} + \beta_j p_j$
endfor
```

Fig. 2.1 *The preconditioned CG algorithm.*

satisfy certain orthogonality conditions. For a symmetric positive definite linear system, these conditions imply that the distance between the approximate solution and the true solution is minimized. For most practical matrices, the SPMV and the triangular solves dominate the other operations when using incomplete factorization as a preconditioner. This is demonstrated by the results given in section 5. Note that both AXPY and DOT are dense operations and hence insensitive to mesh orderings.

2.2. ILU Preconditioning. A preconditioner is any kind of transformation to the original sparse linear system which makes it easier to solve. One broad class of effective preconditioners is based on incomplete factorizations of the matrix A. That is, the preconditioner M is given in the factored form $M = LU$, with L and U being lower and upper triangular matrices. Since some fill elements are suppressed during the factorization process, M is called an ILU preconditioner. Solving with M involves two triangular solutions. In this paper, we consider the simplest form of incomplete factorization, called ILU(0), where all the fill elements not at the nonzero positions of A are discarded. Compared with the other ILU variants, ILU(0) is computationally fast and memory efficient. It is quite effective for a reasonable number of practical matrices. In our implementations, we do not exploit the symmetry of A.

The ILU(0) method contains two steps. First, an incomplete LU factorization of A must be created. This factorization is formally done in place and performed only once, hence its cost is usually negligible compared to the time needed to solve the system.

Second, the lower and upper triangular solves with L and U must be performed in each PCG iteration. The triangular solves incur about the same number of operations as the SPMV Ax, because the sparsity patterns of L and U are identical to the lower and upper triangular parts of A. However, a parallel triangular solve tends to be slower than a parallel SPMV, because it has a smaller degree of parallelism. For SPMV, all the components can be obtained independently in parallel. This is not true for a triangular solve. Figure 2.2 illustrates the lower triangular solve $Lx = b$. The solution of x_i depends on all x_j, $j < i$, unless $l_{ij} = 0$. Thus there are more task dependencies than SPMV, even if L is very sparse. The task dependency graphs change with the matrix ordering; hence, different orderings have different degrees of parallelism. In section 5, we evaluate the performance of PCG on several architectures, using various ordering strategies. Note that the quality of an ILU preconditioner (in terms of its convergence rate) also has a nontrivial dependence on the ordering; however, this is outside the scope of our paper.

```
$x = b$
**for** $j = 1, n$
    $x_j = x_j / l_{jj}$
    **for each** $i > j$ **and** $l_{ij} \neq 0$
        $x_i = x_i - l_{ij} x_j$
**endfor**
```

Fig. 2.2 *The lower triangular solve.*

2.3. Sparse Matrix-Vector Multiplication. The basic SPMV is one of the most heavily used kernels in large-scale numerical simulations, particularly in iterative solution schemes for sparse linear systems. Suppose the coefficient matrix A is of order n and has nnz nonzeros. Then, one SPMV involves $O(nnz)$ floating-point operations, while AXPY and DOT require only $O(n)$ floating-point operations. Thus, SPMV dominates the operation count in CG. To perform the SPMV Ax, we assume that the nonzeros of matrix A are stored in the compressed row storage (CRS) format [1]. The dense vector x is stored sequentially in memory with unit stride. Various numberings of the mesh elements/vertices result in different nonzero patterns of A which, in turn, cause different access patterns for the entries of x. Moreover, on a distributed-memory machine, they imply different amounts of communication.

3. Partitioning and Linearization. Almost all state-of-the-art computer architectures utilize some degree of memory hierarchy (registers, cache, main memory), thus implying that data locality is crucial. With graph partitioning, data locality is enforced by minimizing interprocessor communication, but not at the cache level. However, while graph partitioners are indispensable on distributed-memory machines, they can also be quite useful on shared-memory architectures. In this paper, we have used the METIS [12] multilevel partitioner for our experiments.

Serialization techniques play an important role in enhancing cache performance. Over the years, special numbering strategies (e.g., Cuthill–McKee [4], frontal methods, spectral orderings) have been developed to optimize memory usage and locality of sparse matrix computations. In addition, the runtime support for decomposing adaptive structured grids is often based on a linear representation of the grid hierarchy in the form of a space-filling curve (SFC). SFCs have been demonstrated to be an elegant and unified linearization approach for certain problems in N-body and finite element method (FEM) simulations, mesh partitioning, and other graph-related areas [6, 16, 17, 18, 20]. For our experiments, we pursued both these strategies with some modifications as described below.

3.1. METIS Graph Partitioning. Some excellent parallel graph partitioning algorithms have been developed and implemented in the last decade that are extremely fast, while giving good load balance quality and low edge cuts. Perhaps the most popular is METIS [12], which belongs to the class of multilevel partitioners. METIS reduces the size of the graph by collapsing vertices and edges using a heavy edge matching scheme, applies a greedy graph growing algorithm for partitioning the coarsest graph, and then uncoarsens it back using a combination of boundary greedy and Kernighan–Lin refinement to construct a partitioning for the original graph. Partitioners strive to balance the computational workload among processors while reducing interprocessor communication. Improving cache performance is not a typical objective of most partitioning algorithms.

3.2. RCM Ordering. The particular enumeration of the vertices in an FEM discretization controls, to a large extent, the sparsity pattern of the resulting stiffness matrix. The bandwidth, or profile, of the matrix can have a significant impact on the efficiency of linear systems and eigensolvers.[1] Cuthill and McKee [4] suggested a simple algorithm, called CM, based on ideas from graph theory. Starting from a pseudoperipheral vertex, levels of increasing distance from that vertex are first constructed. The enumeration is then performed level-by-level with increasing vertex degree (within each level). Several variations of this method have been suggested, the most popular being reverse Cuthill–McKee (RCM) [5], where the CM ordering is reversed. In many cases, it has been shown that RCM improves the profile of the resulting matrix. The CM algorithms are fairly straightforward to implement and largely benefit by operating on a pure graph structure; i.e., the underlying graph is not necessarily derived from a triangular mesh. It is worth noting that RCM (or CM) has been observed empirically to be an ordering that is usually good for convergence of ILU.

3.3. SAW Ordering. The general idea of an SFC is to linearize points/elements in a higher dimensional space. This mapping onto a one-dimensional structure is exploited in two ways: First, the locality-preserving nature of the construction fits elegantly into a given memory hierarchy; and second, the partitioning of a contiguous linear object is trivial. For unstructured meshes, an SFC introduces an artificial structure in that the construction depends on the embedding, thereby ignoring the combinatorial structure of the mesh, which drives the formulation of operators between finite element spaces. To overcome this drawback, a novel approach, called a self-avoiding walk (SAW) [7], has recently been proposed. SAW uses a mesh-based (as opposed to geometry-based) technique with similar application areas as SFCs.

A SAW over a triangular mesh is an enumeration of all the triangles such that two consecutive triangles (in the SAW) share an edge or a vertex; i.e., there are no jumps in the SAW. In other words, a SAW visits each triangle exactly once, entering it over an edge or a vertex, and exiting over another edge or vertex. When a SAW goes over vertices, it indicates that the triangles following one another in the enumeration do not share an edge. It is important to note that a SAW is not a Hamiltonian path; however, Hamiltonicity of the dual graph implies the existence of a SAW that goes only over edges [7]. Figure 3.1 shows an example of a SAW over a 36-element triangular mesh.

It has been shown that there is an algorithm for SAW construction whose complexity is linear in the number of triangles in the mesh [7]. Furthermore, SAWs are amenable to hierarchical coarsening and refinement; i.e., they have to be rebuilt only in regions where mesh adaptation occurs and can therefore be easily parallelized. SAW, unlike RCM, is not a technique designed specifically for vertex enumeration; thus, it cannot operate on the bare graph structure of a triangular mesh. This implies a higher construction cost for SAWs, but several different vertex enumerations can be derived from a given SAW.

[1]Sparse direct solvers nowadays usually use minimum degree or nested dissection orderings, which result in much larger bandwidth. However, they preserve sparsity (reduce fill-ins) much better, and run faster. So, although the bandwidth does not impact the efficiency of *all* linear solvers, it does impact the efficiency of CG/PCG iterative solvers that we are examining in this paper.

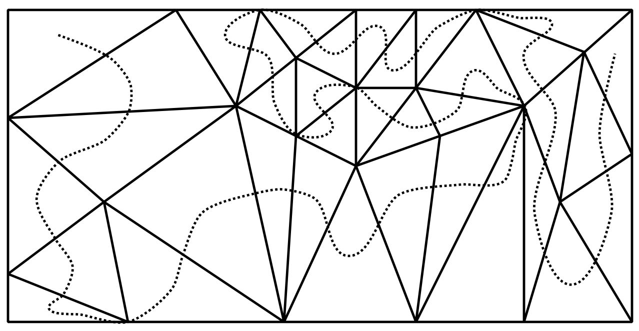

Fig. 3.1 *An example of a SAW over a 36-element triangular mesh.*

4. Programming Paradigms. Recently, four different parallel architectures have emerged, each with its own set of programming paradigms. This work investigates the performance and the programming effort for the CG iterative solver for sparse matrices on each of these architectural platforms using their corresponding programming approaches: message passing, shared-memory directives, hybrid programming, and multithreading. We give below a brief description of these parallel machines and their programming paradigms.

4.1. Message Passing. Parallel programming with message passing is the most common and mature approach for high-performance parallel systems. On distributed-memory architectures, each processor has its own local memory that only it can directly access. To access the memory of another processor, a copy of the desired data must be explicitly sent across the network using a message-passing library such as MPI. To run a code on such machines, the programmer must decide how the data should be distributed among the local memories, communicated between processors during the course of the computation, and reshuffled when necessary. This model causes increased code complexity, especially for irregularly structured applications; however, the benefits lie in enhanced performance for coarse-grained communication and implicit synchronization through blocking communication.

The message-passing experiments in this paper were performed on the distributed-memory architecture of the 640-node Cray T3E, located in the NERSC division of Lawrence Berkeley National Laboratory. Each T3E node consists of a 450 MHz DEC Alpha processor (900 Mflops peak theoretical floating-point speed), 256 MB of main memory, and a 96 KB secondary cache, and is interconnected to other nodes through a three-dimensional torus.

4.2. Shared Memory. Using a shared-memory system can greatly simplify the programming task compared to message-passing implementations. In distributed shared-memory architectures, each processor has a local memory but also has direct access to all the memory in the system. Parallel programs are relatively easy to implement since each processor has a global view of the entire memory. Parallelism can be achieved by inserting compiler directives into the code to distribute loop iterations among the processors. However, portability may be diminished, and performance may suffer from poor spatial locality of physically distributed shared data.

The shared-memory codes presented here were implemented on the 64-node SGI Origin2000, located in the NAS division of NASA Ames Research Center. Each node of the Origin2000 is a symmetric multiprocessor (SMP) containing two 250 MHz MIPS R10000 processors and 512 MB of local memory. The hardware makes all memory equally accessible from a software standpoint, by sending memory requests through routers located on the nodes. Access time to memory is nonuniform, depending on how far away the memory lies from the processor. The topology of the interconnection network is a hypercube, bounding the maximum number of memory hops by a logarithmic function of the number of processors. Each processor also has a relatively large 4 MB secondary cache, where only it can fetch and store data. If a processor refers to data that is not in cache, there is a delay while a copy of the data is fetched from memory. When a processor modifies a word of data, all other copies of the cache line containing that word are invalidated.

OpenMP is the emerging industry standard for shared-memory programming. The parallelism is mostly expressed by loop-level compiler directives. The syntax is similar to many vendors' native pragma directives, such as SGI's IRIX threads, which we used in this study. Alternatively, we could use the lower level POSIX thread (pthreads) library for shared-memory parallelism.[2] This would give us more control over threads and reduce the thread creation times associated with different OpenMP-enabled loops within a single CG iteration. However, a pthreads implementation would increase code complexity significantly.

4.3. Hybrid Programming. The latest technological advances have allowed increasing numbers of processors to have access to a single memory space in a cost-effective manner. As a result, the latest teraflop-scale parallel architectures contain a large number of networked SMPs. Pure MPI codes should port easily to these systems, since message passing is required among the SMP nodes. However, it is not obvious that message passing within each SMP is the most effective use of the system. A recently proposed programming paradigm combines two layers of parallelism by implementing OpenMP shared-memory codes within each SMP, while using MPI among the SMP nodes. This mixed programming strategy allows codes to potentially benefit from loop-level parallelism in addition to coarse-grained domain-level parallelism. Although the hybrid programming methodology may be the best mapping to the underlying architecture, it remains unclear whether the performance gains of this approach compensate for the increased programming complexity and the potential loss of portability.

Figure 4.1 shows a schematic of this hybrid programming paradigm. Two MPI tasks ($task_1$ and $task_2$) are initiated on processors p_1 and p_2 of a node that contains a total of eight processors. Each MPI task then spawns/forks four OpenMP threads, where each individual thread is assigned to a processor. For example, $task_1$ spawns $thread_{12}$, which is assigned to processor p_3. Note that multiple threads can be run on one processor; however, this did not occur for the experiments reported in this paper. Threads spawned by the same task communicate implicitly through shared memory using OpenMP constructs. But threads that were spawned by different tasks communicate explicitly using MPI.

The hybrid architecture used in our experiments is the IBM SP system, recently installed at the San Diego Supercomputing Center (SDSC). The machine contains 1,152 processors arranged as 144 SMP compute nodes. Each node is equipped with 4 GB of memory shared among its eight 222 MHz Power3 processors, and connected via a crossbar. The crossbar technology reduces bandwidth contention to main mem-

[2]In fact, on many systems, OpenMP is implemented based on pthreads.

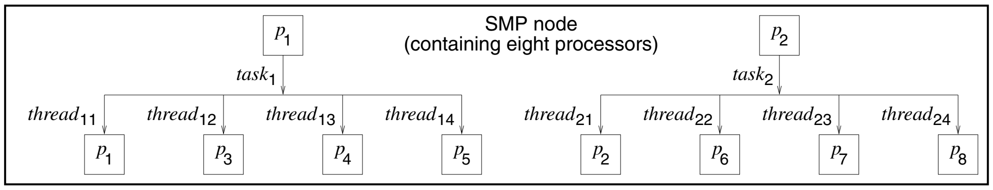

Fig. 4.1 *A schematic of hybrid programming where an SMP node containing eight processors runs two MPI tasks and four OpenMP threads per task.*

ory, compared to traditional shared-bus designs. Each Power3 CPU has an L1 (64 KB) cache which is 128-way set associative and an L2 (4 MB) cache which is four-way set associative with its own private cache bus. All the nodes are connected to each other via a switch interconnect using an omega-type topology. Currently, only four MPI tasks (out of the eight processors) are available within each SMP when using this fast switch. Thus, under the current configuration, the user is required to implement mixed-mode programs to utilize all the processors. The next-generation switch will alleviate this problem.

Hybrid programming may offer an advantage in systems where the MPI library is unoptimized due to software issues (such as a poorly implemented communication layer within an SMP) or hardware limitations (such as the current configuration of the SP's switch at SDSC). Hybrid codes may also benefit applications that are well suited to taking advantage of shared-memory algorithms.

4.4. MTA Multithreading. Multithreading has received considerable attention over the years as a promising way to hide memory latency in high-performance computers, while providing access to a large and uniform shared memory. Using multithreading to build commercial parallel computers is a new concept in contrast to the standard single-threaded microprocessors of traditional supercomputers. Such machines can potentially utilize substantially more of their processing power by tolerating memory latency and using low-level synchronization directives. Cray (formerly Tera) has designed and built a state-of-the-art multithreaded supercomputer called the MTA, which is especially well suited for irregular and dynamic applications. Parallel programmability is considerably simplified since the user has a global view of the memory and need not be concerned with the data layout. However, portability is clearly compromised, as the MTA is currently the only architecture that directly supports this programming paradigm.

The MTA was installed at SDSC in 1998. It has a radically different architecture than current high-performance computer systems. Each 255 MHz processor has support for 128 hardware streams, where every stream includes a program counter and a set of 32 registers. One program thread can be assigned to each stream. The processor switches with no overhead among the active streams at every clock tick, even if a thread is not blocked, while executing a pipelined instruction.

The uniform shared memory of the MTA is flat and physically distributed across hundreds of banks that are connected through a three-dimensional toroidal network to the processors. All memory addresses are hashed by the hardware so that apparently adjacent words are actually distributed across different memory banks. Because of the hashing scheme, it is impossible for the programmer to control data placement. This enhances programmability compared to standard cache-based multiprocessor systems.

Rather than using data caches to hide latency, the MTA processors use multithreading to tolerate latency. If a thread is waiting for its memory reference to complete, the processor executes instructions from other threads. Performance thus depends on having a large number of concurrent computation threads.

Lightweight synchronization among the threads is provided by the memory itself. Each word of physical memory contains a full-empty bit, which enables fast synchronization via load and store instructions without operating system intervention. Synchronization among threads may stall one of the threads but not the processor on which the threads are running, since each processor may run many threads. Explicit load balancing across loops is also not required since the dynamic scheduling of work to threads provides the ability of keeping the processors saturated, even if different iterations require varying amounts of time to complete. Once a code has been written in the multithreaded model, no additional work is required to run it on multiple processors, since there is no difference between uni- and multiprocessor parallelism.

5. Experimental Results.

Our experimental test mesh consists of a two-dimensional Delaunay triangulation, generated by the Triangle [22] software package. The mesh is shaped like the letter "A" and contains 661,054 vertices and 1,313,099 triangles. The underlying matrix is assembled by assigning a random value in $(0, 1)$ to each (i, j) entry corresponding to the vertex pair (v_i, v_j), where $1 \leq distance(v_i, v_j) \leq 3$. The *distance* between two vertices is defined to be the number of edges on the shortest path between them. All other off-diagonal entries are set to zero. This simulates a local discrete operator where each vertex needs to communicate with its neighbors that are no more than three edge lengths away. The matrix is symmetric with its diagonal entries set to 40, which makes it diagonally dominant (and hence positive definite). This ensures that the CG algorithm converges successfully. The final sparse matrix A has approximately 39 entries per row and a total of 25,753,034 nonzeros. This sparsity is representative of matrices obtained from discretizing PDEs on three-dimensional meshes; however, the connectivity pattern will be different for three-dimensional problems. The CG algorithm converges in exactly 13 iterations (tolerance set to 10^{-15}), with the unit vector as the right-hand side b and the zero vector as the initial guess for x. For our test matrix, the SPMV computation accounts for approximately 87% of the total number of floating-point operations within each CG iteration.

For the PCG experiments, the diagonal entries of the matrix were reduced to 10, thus making it no longer diagonally dominant and causing the original CG to fail. However, the ILU(0) PCG algorithm successfully converged in exactly 18 iterations (tolerance again set to 10^{-15}), given the modified matrix.

5.1. Message-Passing Implementation.

In our experiments on the Cray T3E, we use the parallel SPMV and CG routines in Aztec [9], implemented using MPI. The matrix A is partitioned into blocks of rows, with each block assigned to one processor. The associated components of vectors x and b are distributed accordingly. Communication may be needed to transfer some components of x. For example, in $y \leftarrow Ax$, if y_i is updated on processor p_1, $A_{ij} \neq 0$, and x_j is owned by processor p_2, then p_2 must send x_j to p_1. In general, a processor may need more than one x-component from another processor. It is thus more efficient to combine several x-components into one message so that each processor sends no more than one message to another processor. This type of optimization can be performed in a preprocessing phase. The other two operations, AXPY and DOT in the CG algorithm, are easily parallelized: AXPY requires only local computations, whereas DOT requires a local sum followed by a global sum reduction.

Table 5.1 *Runtimes (in seconds) of* `AZ_matvec_mult` *(SPMV) and* `AZ_cg` *(CG) using different strategies on the T3E.*

	ORIG		METIS		RCM		SAW	
P	SPMV	CG	SPMV	CG	SPMV	CG	SPMV	CG
8	0.562	8.652	0.476	7.662	0.381	6.185	0.171	2.916
16	0.325	5.093	0.268	2.909	0.193	3.198	0.086	1.491
32	0.199	3.167	0.087	1.468	0.095	1.662	0.044	0.795
64	0.119	1.929	0.056	0.961	0.045	0.882	0.028	0.462

Table 5.2 *Runtimes (in seconds) of* `AZ_transform` *(initialization) on the T3E.*

P	ORIG	METIS	RCM	SAW
8	504.2	2.829	2.370	2.023
16	547.9	1.455	1.330	1.157
32	333.7	0.840	0.864	0.804
64	150.0	0.422	0.776	0.537

Three routines within Aztec are of particular interest to us: `AZ_transform`, which initializes the data structures and the communication schedule for SPMV, `AZ_matvec_mult`, which performs the matrix–vector multiply, and `AZ_cg`, which solves a linear system using the CG algorithm. In Table 5.1, we report the runtimes of the `AZ_matvec_mult` and `AZ_cg` routines on the T3E at NERSC. It was not possible to run our test problem on fewer than eight processors of this machine due to memory constraints.

Results show that for the key kernel routine `AZ_matvec_mult`, SAW is always about twice as fast as RCM. In turn, RCM is about 1.5 times faster than METIS on 16 or fewer processors, and about the same on 32 or more processors. Note that when using 32 or more processors, METIS is twice as fast as ORIG (the natural ordering from Triangle). For `AZ_cg`, SAW is again about twice as fast as RCM. However, we do not see a clear advantage of RCM over METIS for this routine. Both RCM and METIS are twice as fast as ORIG on large number of processors. Finally, METIS, RCM, and SAW all demonstrate excellent scalability (more than 75% efficiency) up to the 64 processors that were used for these experiments, but ORIG seems less scalable (only about 56% efficiency). As expected, there is a strong correlation between the performance of CG and the underlying SPMV for all test cases.

Table 5.2 shows the preprocessing times spent in `AZ_transform`. The times for METIS, RCM, and SAW are comparable and are usually an order of magnitude larger than the corresponding times for `AZ_matvec_mult`. The `AZ_transform` times also show some scalability up to 32 processors. However, for ORIG, the times are two to three orders larger and show very little scalability. Clearly, the ORIG ordering is too inefficient and unacceptable on distributed-memory machines.

The message-passing PCG experiments in this paper use the BlockSolve95 [11] software library, which is used for solving large, sparse linear systems on parallel platforms that support message passing with MPI. Although Aztec is a powerful iterative library, it does not provide a global ILU(0) factorization routine. BlockSolve95 uses two matrix reordering schemes to achieve scalable performance. First, the graph is reduced by extracting cliques and identical nodes (i-nodes) in the sparse matrix structure, allowing for the use of higher level basic linear algebra subroutines (BLAS). Next, the reduced graph is colored using an efficient parallel coloring heuristic [10]. Finally,

Table 5.3 *Runtimes (in seconds) of* `BStri_solve` *(triangular solve) and* `BSpar_solve` *(PCG) on the SP.*

	ORIG		METIS		RCM		SAW	
P	TriSolve	PCG	TriSolve	PCG	TriSolve	PCG	TriSolve	PCG
8	26.38	96.41	16.95	22.44	10.28	14.63	8.17	11.60
16	16.27	64.23	6.81	9.67	5.10	7.44	4.70	7.29
32	8.95	14.75	4.30	5.97	2.59	4.14	2.35	3.88
64	10.93	15.11	2.63	3.67	1.35	2.36	1.70	2.46

Table 5.4 *Runtimes (in seconds) for* `BSmain_perm` *(graph coloring) and* `BSfactor` *(matrix factorization) on the SP.*

	ORIG		METIS		RCM		SAW	
P	Color	Factorize	Color	Factorize	Color	Factorize	Color	Factorize
8	116.68	339.94	48.41	107.20	37.87	82.53	33.93	75.31
16	75.63	283.71	20.00	46.90	19.01	40.02	17.05	37.22
32	46.96	128.30	10.01	23.26	9.59	20.19	8.76	19.79
64	28.08	82.63	5.01	11.41	5.39	10.48	4.64	9.57

vertices of the same color are grouped and ordered sequentially. As a result, during the triangular solves of the PCG, the unknowns corresponding to these vertices can be solved for in parallel, after the updates from previous color groups have been performed. The number of colors therefore determines the number of parallel steps in the triangular solve. Since BlockSolve95 reorders the input matrix, we investigate what effect, if any, our ordering strategies have on the parallel performance of PCG.

We could not port BlockSolve95 to the T3E because of the large number of MPI tags it requires; hence, our message-passing PCG experiments were conducted on the IBM SP machine. Table 5.3 presents the runtimes of the BlockSolve95 triangular solve (`BStri_solve`) and the total PCG (`BSpar_solve`) routines using various partitioning and ordering strategies. Results clearly show that the initial ordering of the matrix plays a significant role in PCG performance, even though the input matrix is further reordered by the BlockSolve95 library. Notice that `BStri_solve` is responsible for the majority of PCG's computational overhead and is also sensitive to the initial ordering. A comparison of the `BSpar_solve` times shows that RCM and SAW have an advantage over METIS; however, all three schemes are about an order of magnitude faster than ORIG.

Timings for the BlockSolve95 graph coloring (`BSmain_perm`) and ILU(0) matrix factorization (`BSfactor`) routines are presented in Table 5.4. The initial ordering of the matrix dramatically affects both these preprocessing steps, with SAW producing the best results. Notice from Tables 5.3 and 5.4 that the BlockSolve95 library shows scalable performance across all aspects of the PCG computation when intelligent partitioning and ordering schemes are used.

To better understand the various partitioning and ordering algorithms, we have built a simple performance model to predict the parallel runtime of `AZ_matvec_mult`. First, using the T3E's hardware performance monitor, we collected the average number of cache misses per processor. This is reported in Table 5.5 and shows that SAW has the fewest cache misses. In comparison, RCM, METIS, and ORIG have between two and three times that number. Second, we gathered statistics on the average communication volume and the maximum number of messages per processor, both of

Table 5.5 *Locality and communication statistics for* `AZ_matvec_mult` *(SPMV) on the T3E.*

	Avg. Cache Misses ($\times 10^6$)				Avg. Comm. Vol. (in Mbytes) (Max. # Msgs)			
P	ORIG	METIS	RCM	SAW	ORIG	METIS	RCM	SAW
8	3.684	3.034	3.749	2.004	3.228 (7)	0.011 (3)	0.031 (2)	0.049 (6)
16	2.007	1.330	1.905	0.971	2.364 (15)	0.011 (4)	0.032 (2)	0.036 (9)
32	1.060	0.658	1.017	0.507	1.492 (31)	0.009 (5)	0.032 (2)	0.030 (11)
64	0.601	0.358	0.515	0.290	0.828 (63)	0.008 (6)	0.032 (2)	0.023 (16)

Table 5.6 *Predicted runtimes (in seconds) for* `AZ_matvec_mult` *(SPMV) on the T3E. The fraction of the total time spent servicing cache misses is also shown. In the column of total time T, the percentage deviation from the measured time is given in parentheses.*

	ORIG		METIS		RCM		SAW	
P	T (dev.)	$\frac{T_m}{T}$	T (dev.)	$\frac{T_m}{T}$	T (dev.)	$\frac{T_m}{T}$	T (dev.)	$\frac{T_m}{T}$
8	0.367 (-35%)	0.80	0.250 (-47%)	0.97	0.308 (-19%)	0.97	0.169 (-1%)	0.95
16	0.212 (-35%)	0.76	0.110 (-58%)	0.96	0.157 (-19%)	0.97	0.082 (-5%)	0.94
32	0.117 (-41%)	0.72	0.055 (-37%)	0.96	0.084 (-12%)	0.97	0.043 (-2%)	0.94
64	0.067 (-44%)	0.72	0.030 (-46%)	0.96	0.043 (-5%)	0.96	0.025 (-12%)	0.93

which are also shown in Table 5.5. Notice that METIS transfers the least amount of data, whereas RCM has the fewest messages.

In our model, we estimate the total parallel runtime T as

$$T = T_f + T_m + T_c \ ,$$

where T_f, T_m, and T_c are the estimated per-processor times to perform floating-point operations, to service the cache misses, and to communicate the x vector. Given that a floating-point operation requires 1/900 microseconds and that each cache miss latency is 0.08 microseconds (both from T3E product documentation), and assuming that the MPI bandwidth and latency are 50 MB/second and 10 microseconds (both from measurement), respectively, we can estimate the total runtime based on the information in Table 5.5.

Table 5.6 shows the predicted total time T and the ratio T_m/T. T_f is comparatively negligible (consistently less than 5% of T) for all ordering strategies and processor sets. T_c is 18% to 27% of T for ORIG, but less than 3% of T for METIS, RCM, and SAW. In parentheses, we also give the percentage deviation of T from the measured experimental runtime (reported in Table 5.1). The maximum deviation from the measured runtimes is -58%, which gives us some degree of confidence in our model. The results in Table 5.6 clearly indicate that servicing the cache misses is extremely expensive and requires more than 93% of the total time for METIS, RCM, and SAW, and 72% to 80% for ORIG (which has relatively more communication). Although SAW and RCM both incur more communication than METIS (in terms of the average message volume as shown in Table 5.5), their total runtimes are significantly smaller. This illustrates that for our combination of applications and architectures, improving cache reuse can be more important than reducing interprocessor communication.

5.2. Shared-Memory Implementation. This version of the parallel CG code was written using SGI's native pragma directives, which create IRIX threads. A rewrite to OpenMP would require minimal programming effort but has not yet been

Table 5.7 *Runtimes (in seconds) of CG for different orderings running in FLATMEM and CC-NUMA modes on the Origin2000. The CG runtimes for an MPI implementation on the Origin2000 with the SAW ordering are also given for comparison.*

	FLATMEM			CC-NUMA			MPI
P	ORIG	RCM	SAW	ORIG	RCM	SAW	SAW
1	46.911	37.183	36.791	46.911	37.183	36.791	
2	28.055	21.867	21.772	27.053	21.454	21.229	23.145
4	30.637	25.350	24.751	17.608	10.651	10.593	7.880
8	16.836	14.431	14.121	9.824	5.575	5.516	3.815
16	16.348	15.516	15.548	6.205	2.845	2.872	1.926
32	16.653	15.350	15.423	3.584	1.548	1.514	1.075
64	10.809	7.782	8.450	2.365	0.885	0.848	0.905

done at this time. Each processor is assigned an equal number of rows in the matrix. The parallel SPMV and AXPY routines do not require explicit synchronizations, since they do not contain concurrent writes. Global reduction operations are required for DOT and the convergence tests. Two basic implementation approaches described below were taken.

The FLATMEM strategy naively assumes that the Origin2000 is a flat shared-memory machine. Arrays are not explicitly distributed among the processors, and nonlocal data requests are handled by the cache coherent hardware. Alternatively, the CC-NUMA strategy addresses the underlying distributed-memory nature of the machine by performing an initial data distribution. Sections of the sparse matrix are appropriately mapped onto the memories of their corresponding processors using the default "first touch" data distribution policy of the Origin2000. The computational kernels of both the FLATMEM and CC-NUMA implementations are identical, and simpler to implement than the MPI version. Table 5.7 shows the SPMV and CG runtimes using both approaches with the ORIG, RCM, and SAW orderings of the mesh. We also present CG runtimes using an MPI implementation on the Origin2000 with the SAW ordering as a basis for comparison.

Observe that the CC-NUMA implementation shows significant performance gains over FLATMEM. This is expected since the Origin2000 is a distributed-memory system and therefore should be treated as such. As the number of processors increases, the runtime difference between the two approaches becomes more dramatic, achieving an order of magnitude improvement when using more than 16 processors. Proper data distribution becomes increasingly important for larger numbers of processors since the corresponding communication overhead grows nonuniformly. Within the CC-NUMA approach, the RCM and SAW ordering schemes dramatically reduce the runtimes compared to ORIG, indicating that an intelligent ordering algorithm is necessary to achieve good performance and scalability on distributed shared-memory systems. There is little difference in parallel performance between RCM and SAW because both ordering techniques reduce the number of secondary cache misses and the nonlocal memory references of the processors. Recall, however, that on the T3E, SAW was about twice as fast as RCM. This discrepancy in performance is probably due to the larger cache size of the Origin2000, which reduces the individual effects of the two ordering strategies.

The last two columns of Table 5.7 compare the CC-NUMA and MPI implementations of CG on the Origin2000 using the SAW ordering. Notice that the runtimes are very similar, even though the programming methodologies of these two approaches are quite different. These results indicate that for this class of applications, it is possible to achieve message-passing performance using shared-memory constructs through careful data ordering and distribution.

A shared-memory version of PCG is currently unavailable and is not considered in this paper. An efficient implementation would require a CC-NUMA approach with similar algorithmic designs to those used in the BlockSolve95 library, including graph dependency analysis and matrix reordering. As we have shown in this section and in previous work [15, 21], a simplified FLATMEM strategy produces poor results for irregularly structured problems and would not be suitable for PCG.

5.3. Hybrid Implementation. For the hybrid implementation of the CG algorithm on the IBM SP, we started with the Aztec MPI library [9] and incrementally added OpenMP parallelization directives. Through the use of profiling, the key loop nests responsible for significant portions of the overall execution were identified. A naive parallelization of all loops can be counterproductive since the overhead of OpenMP can exceed the savings in execution time. Some reorganization of the code, including the use of temporary variables, was necessary to preserve correctness. In all, eight Aztec loops were parallelized with OpenMP directives, the most important being the SPMV routine. To achieve the best possible OpenMP performance, dense vector operations were performed with the threaded vendor-optimized BLAS from IBM's Engineering Scientific Subroutine Library (ESSL).

Table 5.8 shows the results of the hybrid CG implementation on the SP for varying numbers of SMP nodes, MPI tasks, and OpenMP threads. In addition to the ORIG, METIS, RCM, and SAW orderings, we present a new hybrid partitioning/linearization scheme comprised of METIS+SAW. Since METIS [12] is well suited for minimizing interprocessor communication and SAW [7] has been demonstrated to enhance cache locality, combining these two approaches is a potentially promising strategy for hybrid architectures. First, the graph is partitioned into the appropriate number of MPI tasks using METIS. Next, a SAW linearization is applied to each individual subdomain in parallel. Thus, when multiple OpenMP threads process their assigned submatrix, the SAW reordering should improve each processor's cache performance and reduce false sharing.

Notice that when there is only one SMP node and one MPI task (as in {1,1,4} and {1,1,8}),[3] the CG code is effectively parallelized using only OpenMP; thus, timings are not presented for the corresponding METIS and METIS+SAW entries. The performance trends are very similar to the CC-NUMA results in section 5.2 where the SAW ordering gave the best runtimes. Similarly, when the number of OpenMP threads is one, the parallelization is purely MPI based. Recall from section 4.3 that due to limitations in the current switch architecture of SDSC's SP, the maximum number of MPI tasks is limited to four on each SMP, and hybrid programming is required to use all the available processors.

The performance of the ordering schemes averaged across all combinations of nodes, tasks, and threads from best to worst are: METIS+SAW, SAW, RCM, METIS, and ORIG. The METIS+SAW strategy consistently outperforms all others; however, as was shown in section 5.1, cache behavior is significantly more important than interprocessor communication for our application. As a result, there is no significant performance difference between the hybrid METIS+SAW strategy and the pure SAW linearization. Nonetheless, we expect algorithms with higher communication

[3]The tuple $\{x, y, z\}$ denotes {SMP nodes, MPI tasks, OpenMP threads}.

Table 5.8 *Runtimes (in seconds) of CG using different orderings on the SP.*

P	Nodes	Tasks	Threads	ORIG	METIS	RCM	SAW	METIS+SAW
4	1	1	4	6.470		3.561	3.244	
	1	2	2	6.848	4.968	3.294	3.017	2.990
	1	4	1	7.262	3.994	3.192	2.919	2.905
	2	1	2	6.928	4.804	3.255	2.962	2.921
	2	2	1	7.656	3.881	3.136	2.829	2.805
	4	1	1	7.278	3.871	3.108	2.803	2.772
8	1	1	8	4.388		2.162	1.998	
	1	2	4	4.995	2.929	1.992	1.879	1.841
	1	4	2	6.038	2.426	1.930	1.812	1.781
	2	1	4	4.858	2.768	1.858	1.716	1.675
	2	2	2	5.955	2.234	1.759	1.620	1.589
	2	4	1	6.141	1.891	1.758	1.595	1.575
	4	1	2	5.301	2.123	1.733	1.568	1.530
	4	2	1	6.044	1.806	1.687	1.506	1.494
	8	1	1	5.550	1.774	1.687	1.511	1.453
16	2	1	8	3.375	1.926	1.217	1.139	1.118
	2	2	4	4.125	1.366	1.071	1.018	0.992
	2	4	2	4.782	1.472	1.084	1.019	1.006
	4	1	4	3.684	1.261	0.987	0.925	0.897
	4	2	2	4.527	1.078	0.985	0.894	0.884
	4	4	1	5.186	0.965	0.986	0.914	0.902
	8	1	2	4.153	1.057	0.960	0.892	0.847
	8	2	1	4.539	0.905	0.926	0.842	0.828
32	4	1	8	2.973	0.870	0.651	0.678	0.617
	4	2	4	3.608	0.709	0.618	0.593	0.581
	4	4	2	4.067	0.723	1.120	0.680	0.649
	8	1	4	3.325	0.628	0.590	0.529	0.506
	8	2	2	3.801	0.587	0.592	0.560	0.545
	8	4	1	4.267	0.586	0.607	0.580	0.569
64	8	1	8	2.992	0.473	0.391	0.390	0.372
	8	2	4	3.557	0.452	0.690	0.442	0.407
	8	4	2	3.963	0.466	0.798	0.495	0.460

requirements to benefit from this dual partitioning/ordering approach. This will be the subject of future research. Overall, these results show that intelligent ordering schemes are extremely important for efficient sparse matrix computations regardless of whether the programming paradigm is OpenMP, MPI, or a combination of both.

To compare hybrid versus pure MPI performance, first examine the METIS+SAW column since it gives the best CG runtimes. Each processor set shows differing results. For example, on 16 processors, the fastest CG implementation is for {8,2,1}, meaning no OpenMP parallelization is triggered. However, on 32 processors, {8,1,4} is the fastest, outperforming {8,4,1}. Finally, on 64 processors, using the maximum number of OpenMP threads, as in {8,1,8}, gives the best results. Within each processor set, varying the number of tasks and threads does not result in a significant performance difference. Overall, the hybrid implementation offers no noticeable advantage. This is true for the other ordering schemes as well, as is evident from Table 5.8. However, since the hybrid paradigm increases programming complexity and adversely affects portability, we conclude that for running iterative sparse solvers on clusters of SMPs, a pure MPI implementation is a more effective strategy. Similar conclusions have been drawn in recent work by other researchers [3, 8].

For the same reasons as mentioned in section 5.2, a hybrid PCG implementation is not considered in this paper and will be the subject of future work.

Table 5.9 *Runtimes (in seconds) for the original and SAW orderings on the MTA.*

	ORIG		SAW
P	SPMV	CG	CG
1	0.378	9.86	9.74
2	0.189	5.02	5.01
4	0.095	2.53	2.64
8	0.051	1.35	1.36

The results in sections 5.1, 5.2, and here show that if the underlying computation undergoes dynamic mesh adaptation, a new reordering would be required each time the mesh evolved for efficient parallel performance. In addition, once an ordering was computed for the newly adapted mesh, a remapping phase would be necessary to appropriately redistribute the corresponding submatrix onto the processors. These processes preserve the computational load balance and maintain good cache locality for adaptive applications. Unfortunately, a significant overhead is generally associated with these rebalancing phases [14, 15, 21]. The CC-NUMA and MPI+OpenMP strategies would thus be comparable to an MPI implementation, requiring similar amounts of programming effort and rebalancing overheads. The major difference would be the use of a shared address space (global on an Origin2000, local within an SMP node of an SP) instead of explicit message-passing calls for interprocessor communication.

5.4. MTA Multithreaded Implementation. The multithreaded implementation of CG on the Cray MTA is straightforward, requiring only compiler directives. Since the data structures are dynamically allocated pointers, special pragma assertions were used to indicate that there are no loop-carried dependencies. The compiler was thus able to automatically parallelize the appropriate loop segments. Load balancing is implicitly handled by the operating system, which dynamically assigns rows to threads. The reduction operations for DOT and the convergence test were handled automatically as well. Otherwise, special synchronization constructs were not required since there are no other possible race conditions in the multithreaded CG. It is important to highlight that no special ordering was necessary to achieve good parallel performance.

Results using 60 streams per processor are presented in Table 5.9. Both CG and the underlying SPMV achieve high efficiency of over 90% using the ORIG ordering. This indicates that there is enough thread- and instruction-level parallelism in CG to tolerate the relatively high overhead of memory access. There is a slight drop in performance between four and eight processors. As we increase the number of processors, the number of active threads increases proportionately while the runtimes become very small. As a result, a greater percentage of the overall time is spent on thread management, causing a decrease in efficiency. Notice that the SAW ordering does not significantly change the performance of CG on this cache-less architecture. Thus, the programming and runtime overheads associated with partitioning/linearization schemes are absent on this platform. Furthermore, reordering and remapping are not required even if the underlying mesh undergoes adaptation. This saves both the computational resources and the programming overhead of rebalancing the mesh in an adaptive environment. Thus, the MTA has a distinct advantage over distributed-memory systems for this class of adaptive application.

For the MTA implementation of PCG, we developed a multithreaded version of the lower and upper triangular solves (see Figure 2.2). Matrix factorization times are not reported since it is performed only once outside the inner loop. Our multithreaded

Table 5.10 *Runtimes (in seconds) for the triangular solve and the overall PCG on the MTA.*

	ORIG	
P	TriSolve	PCG
1	71.98	80.34
2	45.74	50.02
4	26.94	29.18
8	16.04	17.29

strategy uses low-level locks to effectively perform an on-the-fly dependency analysis. Recall that to compute the lower triangular solve $Lx = b$, the solution of x_i depends on all x_j, $j < i$, unless $l_{ij} = 0$. First, synchronization locks are applied to all x_j, $j = 1, 2, \ldots, n$, to guarantee correct dependency behavior. Threads are then dynamically assigned to solve for each x_i. If a given x_i has a dependency on x_j that has not yet been computed, the attempt to access the blocked memory address of x_j will cause the thread responsible for processing x_i to be temporarily put to sleep. Once a thread successfully solves for x_j, the synchronization lock on that variable is released, causing the runtime system to wake all blocked threads waiting to access the memory address of x_j. Subsequent attempts to access that variable will no longer cause active threads to become blocked. The lightweight synchronization of the MTA allows locks to be effectively used at such a fine granularity. Notice that the multithreaded version of triangular solve is dramatically less complex than the BlockSolve95 implementation described in section 5.1, which required advanced graph dependency analysis and matrix reordering to achieve high parallelism.

Table 5.10 presents the performance of PCG with the ORIG ordering on the MTA, again using 60 streams. Observe that the triangular solve is responsible for most of the computational overhead and achieves a speedup of approximately 4.5X on eight processors. This limited scalability is due to the lack of available thread-level parallelism in our dynamic dependency scheme. A large fraction of the computational threads were blocked at any given time, preventing a full saturation of the MTA processors. Subsequent attempts to optimize the multithreaded code by increasing the number of streams and using more sophisticated orderings strategies caused the machine to crash due to limitations in its current system software.[4] We plan to revisit the multithreaded PCG once a more mature runtime system becomes available on the MTA. It would also be interesting to continue our experiments as more processors are added to the system.

6. Summary and Conclusions. In this paper, we examined the performance of and the programming effort required for the CG sparse iterative solver on four leading parallel platforms using their corresponding programming approaches: message passing, shared-memory directives, hybrid programming, and multithreading.

Parallel programming with message passing is the most common and mature approach for high-performance systems. The MPI version of CG on the Cray T3E used the Aztec [9] library. We compared the parallel performance after ordering the sparse matrix using reverse Cuthill–McKee (RCM) [4], self-avoiding walk (SAW) [7], and the METIS partitioner [12]. Results showed that all three schemes greatly improve the parallel performance of CG compared to the naive natural ordering. In addition,

[4]We were requested by the MTA system administrator at SDSC to postpone running our PCG code until the system software error could be isolated and corrected.

we demonstrated that traditional graph partitioners, which focus on minimizing edge cuts, are not necessarily the best tools for distributing sparse matrices on multiprocessor systems. Using RCM or SAW as an ordering (and partitioning) strategy results in a faster CG than METIS, due to better cache reuse. This shows that the cache implications of ordering are more important than the communication implications of partitioning. A performance model was also presented which predicts the expected sparse matrix–vector multiply (SPMV) runtime as a function of both cache misses and communication overhead. Within each CG iteration, the SPMV is usually the most expensive operation.

For ill-conditioned linear systems, it is often necessary to use a preconditioning technique. We presented MPI results for ILU(0) preconditioned CG (PCG) using the BlockSolve95 [11] library. Unlike CG, the runtime of the PCG algorithm is dominated by the triangular solves, which are inherently less amenable to parallelization than SPMV. BlockSolve95 graph colors and reorders the input matrix to achieve high parallelism; however, we found that the initial ordering of the input matrix dramatically affected PCG's performance. Overall, the SAW linearization resulted in the best runtimes for all components of PCG, including graph coloring and factorization.

Using a shared-memory system can greatly simplify the programming task compared to message passing. A shared-memory implementation of CG on the SGI Origin2000 showed that ordering algorithms dramatically improve parallel performance. This is because the Origin2000 is a distributed-memory architecture, so proper data distribution is required even when programming in shared-memory mode. A direct comparison with an MPI implementation indicated that it is possible to achieve message-passing performance using shared-memory constructs for this class of applications through careful data ordering and distribution.

A recently proposed hybrid programming paradigm combines two layers of parallelism by implementing OpenMP shared-memory codes within an SMP while using MPI among the SMP clusters. We developed the CG algorithm on the IBM SP, by starting with the Aztec [9] MPI library and incrementally adding OpenMP parallelization directives. A new hybrid strategy comprised of METIS+SAW was presented and consistently outperformed the other schemes. However, since cache behavior is significantly more important than interprocessor communication for our application, there was little performance difference between the METIS+SAW strategy and pure SAW linearization. Comparing hybrid (MPI+OpenMP) versus pure MPI implementations of CG, we found negligible performance difference between the two schemes. However, since the hybrid paradigm increases programming complexity and adversely affects portability, we conclude that for running iterative solvers on clusters of SMPs, a pure MPI implementation is a more effective strategy.

Multithreading has received considerable attention over the years as a promising way to hide memory latency in high-performance computers while providing access to a large and uniform shared memory. We presented results on the multithreaded architecture of the Cray MTA. The CG implementation was straightforward, requiring only compiler directives. Results showed that special ordering and/or partitioning schemes are not required on the MTA to obtain high efficiency and scalability. Furthermore, reordering and remapping are not required even if the underlying mesh undergoes adaptation, giving the MTA a distinct advantage over distributed-memory systems for adaptive applications. However, portability is lost as the MTA is currently the only architecture that directly supports this programming paradigm.

Finally, a multithreaded version of the PCG algorithm was also developed. Here, the triangular solve uses low-level locks to perform a graph dependency analysis at runtime. This implementation was dramatically less complex than BlockSolve95's PCG, which required advanced graph dependency analysis and matrix reordering. However, only limited scalability was achieved due to the lack of available thread-level parallelism in our dynamic dependency scheme, which prevented a full saturation of the MTA processors.

In conclusion, this paper examined the intricate relationships among ordering and partitioning schemes, parallel programming paradigms, and multiprocessor architectures for a class of sparse matrix computations. We expect that the methodologies and lessons derived from this research will be applicable to other application domains characterized by irregular data access.

Acknowledgments. The authors would like to sincerely thank Lie-Quan (Rich) Lee for implementing the METIS+SAW partitioning/linearization scheme on the SP. Thanks also to Bruce Hendrickson and the anonymous referees for their suggestions that helped improve the paper.

REFERENCES

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for the Iterative Methods*, SIAM, Philadelphia, 1994.

[2] D. A. Burgess and M. B. Giles, *Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines*, Adv. Engrg. Software, 28 (1997), pp. 189–201.

[3] F. Cappello and D. Etiemble, *MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks*, in Proc. Supercomputing'00, Dallas, TX, 2000.

[4] E. Cuthill and J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, in Proceedings of the 24th ACM National Conference, 1969, pp. 157–172.

[5] A. George, *Computer Implementation of the Finite Element Method*, Technical Report STAN-CS-208, Stanford University, Stanford, CA, 1971.

[6] M. Griebel and G. Zumbusch, *Hash-storage techniques for adaptive multilevel solvers and their domain decomposition parallelization*, in Domain Decomposition Methods, 10 (Boulder, CO, 1997), Contemp. Math. 218, AMS, Providence, RI, 1998, pp. 279–286.

[7] G. Heber, R. Biswas, and G. R. Gao, *Self-avoiding walks over adaptive unstructured grids*, Concurrency: Pract. Exper., 12 (2000), pp. 85–109.

[8] D. S. Henty, *Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling*, in Proc. Supercomputing'00, Dallas, TX, 2000.

[9] S. A. Hutchinson, L. V. Prevost, J. N. Shadid, and R. S. Tuminaro, *Aztec User's Guide*, Technical Report SAND95-1559, Sandia National Laboratories, Albuquerque, NM, 1998.

[10] M. T. Jones and P. E. Plassmann, *A parallel graph coloring heuristic*, SIAM J. Sci. Comput., 14 (1993), pp. 654–669.

[11] M. T. Jones and P. E. Plassmann, *BlockSolve95 User's Manual: Scalable Library Software for the Parallel Solution of Sparse Linear Systems*, Technical Report ANL-95/48, Argonne National Laboratory, Chicago, IL, 1995.

[12] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1998), pp. 359–392.

[13] R. Löhner, *Renumbering strategies for unstructured-grid solvers operating on shared-memory, cache-based parallel machines*, Comput. Methods Appl. Mech. Engrg., 163 (1998), pp. 95–109.

[14] L. Oliker and R. Biswas, *PLUM: Parallel load balancing for adaptive unstructured meshes*, J. Parallel Distrib. Comput., 52 (1998), pp. 150–177.

[15] L. Oliker and R. Biswas, *Parallelization of a dynamic unstructured algorithm using three leading programming paradigms*, IEEE Trans. Parallel Distrib. Systems, 11 (2000), pp. 931–940.

[16] C.-W. Ou, S. Ranka, and G. Fox, *Fast and parallel mapping algorithms for irregular problems*, J. Supercomputing, 10 (1995), pp. 119–140.

[17] M. Parashar and J. C. Browne, *On partitioning dynamic adaptive grid hierarchies*, in Proceedings of the 29th Hawaii International Conference on System Sciences, Wailea, HI, 1996, pp. 604–613.

[18] J. R. Pilkington and S. B. Baden, *Dynamic partitioning of non-uniform structured workloads with space-filling curves*, IEEE Trans. Parallel Distrib. Systems, 7 (1996), pp. 288–300.

[19] Y. Saad, *Iterative Methods for Sparse Linear Systems*, PWS, Boston, MA, 1996.

[20] J. Salmon and M. S. Warren, *Parallel, out-of-core methods for fast evaluation of long-range interactions*, in Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, 1997, CD-ROM, SIAM, Philadelphia, 1997.

[21] H. Shan, J. P. Singh, L. Oliker, and R. Biswas, *A comparison of three programming models for adaptive applications on the Origin*2000, J. Parallel Distrib. Comput., 62 (2002), pp. 241–266.

[22] J. R. Shewchuk, *Triangle: Engineering a* 2*D quality mesh generator and Delaunay triangulator*, in Applied Computational Geometry: Towards Geometric Engineering, Lecture Notes in Comput. Sci. 1148, Springer-Verlag, Heidelberg, Germany, 1996, pp. 203–222.